Modelling Hardware Verification Concerns Specified in the *e* Language: An Experience Report

Darren Galpin
Infineon Technologies
Hunts Ground Road,
Stoke Gifford,
Bristol, England, BS34 8HP
+44 (0)117 9528741
Darren.Galpin@infineon.com

Cormac Driver
Lero,
Distributed Systems Group
Trinity College Dublin,
Ireland
+353 1 8961765
Cormac.Driver@cs.tcd.ie

Siobhán Clarke
Lero,
Distributed Systems Group
Trinity College Dublin,
Ireland
+353 1 8962224
Siobhan.Clarke@cs.tcd.ie

ABSTRACT

e is an aspect-oriented hardware verification language that is widely used to verify the design of electronic circuits through the development and execution of testbenches. In recent years, the continued growth of the testbenches developed at Infineon Technologies has resulted in their becoming difficult to understand, maintain and extend. Consequently, a decision was taken to document the testbenches at a higher level of abstraction. Accordingly, we attempted to model our legacy test suites with an existing aspect-oriented modelling approach. In this paper we describe our experience of applying Theme/UML, an aspect-oriented modelling approach, to the representation of aspect-oriented testbenches implemented in *e*. It emerged that the common aspect-oriented concepts supported by Theme/UML were not sufficient to adequately represent the *e* language, primarily due to *e*'s dynamic, temporal nature. Based on this experience we propose a number of requirements that must be addressed before aspect-oriented modelling approaches such as Theme/UML are capable of representing aspect-oriented systems implemented in *e*.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.10 [**Software Engineering**]: Design; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Languages, Verification

Keywords

Hardware verification, *e*, aspect-oriented modelling, Theme/UML

1. INTRODUCTION

Infineon Technologies [1] offers semiconductors and system solutions for automotive, industrial electronics, chip card and security domains, as well as applications in communications. These products are based on development of innovative analog and mixed signal, radio frequency, power and embedded control technologies. The company currently holds approximately 22,900 patents related to these technologies.

Specman is an aspect-oriented hardware verification tool employed by Infineon to verify the design of electronic circuits. Specman testbenches are written in an aspect-oriented language called *e*. *e* is primarily used by Infineon to construct testbenches that inject stimuli into a hardware design and check for valid actions/responses. The continued growth of the testbenches in recent years has raised the issue of how to maintain and reuse legacy test code as it increases in complexity. Testbenches are reused across multiple projects, with different aspect-oriented extensions being added on a per-project basis. When attempting to adopt a legacy testbench written by another developer, the number of aspect-oriented extensions makes it difficult to determine exactly what behaviour has been defined, particularly when individual methods can be extended and overwritten in different sub-types. To compound this problem, knowledge of the particulars of each project-specific design is partially or completely lost as people move out of Infineon. It was decided that using a high-level, graphical design language to visualise the testbenches would speed up knowledge transfer and project understanding, and thereby aid with maintaining and reusing existing code.

UML was initially used to model the testbenches, but we found that the object-oriented concepts embodied in UML did not map to *e*'s aspect-oriented concepts. Subsequently, an aspect-oriented extension to UML was adopted. This paper describes our experience with attempting to model Infineon's testbenches with an established aspect-oriented modelling approach called Theme/UML [2]. We found that while the common aspect-oriented notions supported by Theme/UML were a better fit with those in *e* than the object-oriented notions in UML, there were still a number of issues that prevented us from successfully modelling the testbenches. These issues were primarily related to *e*'s support for dynamism and temporality - concepts not commonly supported in aspect-oriented software development techniques. This situation was exacerbated by a misalignment between *e*'s basic units of decomposition and those generally used in aspect-oriented languages. The result of this experience, and the contribution of this paper, is a set of requirements for future aspect-oriented modelling approaches.

The remainder of this paper is structured as follows. Section 2 introduces the *e* language and discusses its support for aspect-oriented programming. Section 3 discusses our experience with

attempting to model testbenches written in *e* with Theme/UML. Section 4 proposes a set of requirements for aspect-oriented modelling approaches that arose from the work described in Section 3. Section 5 contains a summary.

2. THE *e* PROGRAMMING LANGUAGE

e is a domain-specific programming language that is used for functional verification of electronic designs. The language was developed in 1997 by Verisity Design (subsequently acquired by Cadence Design Systems [3]) as part of their Specman tool and from its inception contained constructs supporting aspect-oriented programming. *e* was standardised as IEEE 1647 [4] and a second revision of the standard was published in 2008. In this section we introduce the *e* language by describing its key concepts and examining its aspect-oriented features.

At *e*'s core is a pseudo-random generator that facilitates creation of input stimuli that can be applied to the design under test (DUT). All variables are assigned a random value unless either marked as not generatable or constrained to be a specific value. *e* contains constructs that allow the response of the DUT to be monitored and checked. In addition, there are constructs to support assessment of the functional coverage of the DUT (as opposed to simply the code coverage).

A typical electronic design consists of a processor, peripherals and connecting busses, all of which are usually verified individually before being tested together. The individual peripherals are verified in their own testbenches using *e*-Verification Components (*e*VCs). *e*VCs are testbench elements written in *e* that encapsulate a protocol or some specific behaviour and are used to stimulate the design, test the response and measure whether certain state combinations and system responses have been observed, i.e., assess the functional coverage. The most common type of *e*VC is one targeted at a specific bus protocol, and contains a bus master to generate stimuli, a bus slave to receive responses from a DUT master, a bus arbiter to control the system, protocol checkers to ensure that the correct bus behaviour is observed at all times, and a scoreboard to check for end-to-end data transmission correctness throughout the system.

In providing support for the development of such testbenches, *e* brings together concepts from several languages:

- It has a basic object-oriented programming model with automatic memory management and single inheritance in a similar manner to Java.
- *e* natively supports aspects that can cut across multiple objects within an *e* domain.
- *e* supports constraints as object features, using constraints to refine object models. The execution model resolves the constraints, picking random values that satisfy the constraint set.
- *e* is strongly typed, like Pascal.
- *e* has concurrency constructs for hierarchical composition, similar to hardware description languages such as Verilog and VHDL.
- *e* contains temporal logic constructs that borrow from linear temporal logic and interval temporal logic.

e supports aspect-oriented programming through two main mechanisms, one for objects and one for methods. Unlike object-oriented languages such as Java, *e* does not have classes, but instead has units and structs. Units represent objects that exist throughout the simulation time, e.g., bus masters and slaves. Structs represent objects that are created and destroyed during simulation time, e.g., bus data packets that are passed between agents in a system.

Both units and structs can have aspect-oriented extensions applied to them in the same manner through the use of the `extend` construct. This is similar to the concept of inter-type declarations in AspectJ. The `extend` construct adds code to an existing module in an aspect-oriented manner (as opposed to creating a new module as per standard inheritance, which is achieved in *e* using the `like` keyword). Listing 1 illustrates a simple example of how a unit and a struct can have aspect-oriented extensions applied to them. An empty object is defined in each case. The base object is extended by adding a new variable to the object. The same mechanism can be used to extend types. Types are used to represent data, are globally visible, and can be user-defined enumerated variables as well as the pre-defined numerical and Boolean types common to most languages. The type extension mechanism is illustrated in Listing 2 where an extra type is added to `dog_type` (which is a collection of possible dog types).

```
unit base_unit { };
extend base_unit {
    a : unit;
};

struct base_struct { };
extend base_struct {
    b : bool;
}
```

Listing 1. Unit and struct extension

```
type dog_type : [poodle, bulldog];
extend dog_type : [pug];
```

Listing 2. Type extension

Method extension is handled via three aspect-oriented constructs, `is also`, `is first`, and `is only`. Consider the following methods:

(a) `bark() is {out("Bark!")};`

(b) `bark() is first {out("1st message")};`

(c) `bark() is also {out("Last message")};`

(d) `bark() is only {out("Miaow!")};`

(a) is the original method declaration and prints the message "Bark!" when executed. When (a) and (b) are deployed together, two messages are printed - first "1st message" and second "Bark!". If (c) is also active then three messages are printed, with "Last message" printed last. However, if (d) is loaded after (a), (b) and (c), the `is only` construct dictates that all previous definitions are ignored and the latest definition becomes the one that is used. In this case the only message printed is "Miaow!".

As illustrated in Listing 2, the original `dog_type` type had two concrete types, and the aspect extension adds a third type to the list. Aspect-oriented method extensions can be applied to both

existing and newly created types. Listing 3 illustrates how different types can have different behaviours based on aspect-oriented method extensions. In this way complex layers of aspects can be introduced to the system to change the observed behaviour.

```
unit dog {
    dog_kind : dog_type;
    bark() is empty;

    when pug dog {bark() is also
        (out("I am a Pug"))};
    when poodle dog {bark() is also
        (out("I am a Poodle"))};
};
```

Listing 3. Aspect extensions within polymorphic extensions

e also supports aspect-oriented extension of temporal events and coverage objects in a similar way to how it handles method extension. A temporal expression is a combination of events and temporal operators that describe behaviour, temporal relationships between events, field values, variables and other items during a test. An event may be emitted during execution of a method either by directly invoking `emit event` or because a signal to which the event is tied has changed. For example, `event a is rise('clk')@sim` is an event that is true if the signal `clk` within the DUT rises on a simulator tick. Because of the use of `is`, an event can be extended using the `is only` extension (`is first` and `is also` are not supported for temporal events). Consequently, a new aspect could be created in order to redefine event `a`, e.g., `event a is only fall('clk')@sim`. This action results in event `a` looking for the falling edge of `clk` at `sim` rather than the rising edge.

Coverage objects, triggered by specific events, are used to collect functional coverage information about key architectural and micro-architectural features of the DUT. Listing 4 illustrates the syntax of a coverage object.

```
cover <event>[using <options>] is [also] {
 item name [:type=expr] [using <options>]
};
```

Listing 4. Coverage object syntax

If a coverage object has been defined, new objects can be added using the `is also` extension. The `<options>` field in the cover object definition allows further constraints to be added. A typical use of this facility is to add constraints that restrict evaluation of the coverage object to times when certain conditions hold. For example, the statement `when=FSM==idle` restricts the evaluation of the coverage object to times when the finite state machine has the value `idle` at the time the coverage event occurs. The options can also be extended, giving the user the ability to extend the coverage object within an aspect, as illustrated by example in Listing 5.

```
cover barking is {
   item dog_kind;
};

cover barking using is also when =
   weight > 50{};
```

Listing 5. Aspect extension of coverage objects

The example shows that the cover group[1] `barking` is triggered when the barking event is observed and the `dog_kind` is recorded. The extension adds a restriction to this, only allowing the evaluation to occur when the variable `weight` is greater than 50. This type of extension can be performed on coverage objects as well as cover groups. The extension differs from the others described previously in that the extension itself can contain a constraint on when it is executed, allowing different aspects of the same object to record different ranges of information. It can only be performed on cover objects.

In the next section we discuss our experience with attempting to model testbenches written in *e* with Theme/UML, a UML-based aspect-oriented modelling language.

3. MODELLING HARDWARE VERIFICATION CONCERNS SPECIFIED IN *e*

UML is a general-purpose, language-independent modelling approach that is widely used to design object-oriented systems. We initially attempted to model our testbenches using UML but found that it lacked support for the aspect-oriented features of *e*. We subsequently investigated existing aspect-oriented modelling approaches and decided to attempt to model our testbenches with Theme/UML. Theme/UML is an aspect-oriented extension to UML that supports fine-grained decomposition and composition of both functional and non-functional concerns, including those that are crosscutting. Theme/UML proved a better fit than standard UML to the aspect-oriented constructs in *e*. However, there were a number of instances in which Theme/UML could not adequately capture behaviour that can be specified with *e*. These issues arose primarily due to the dynamic, temporal nature of *e*.

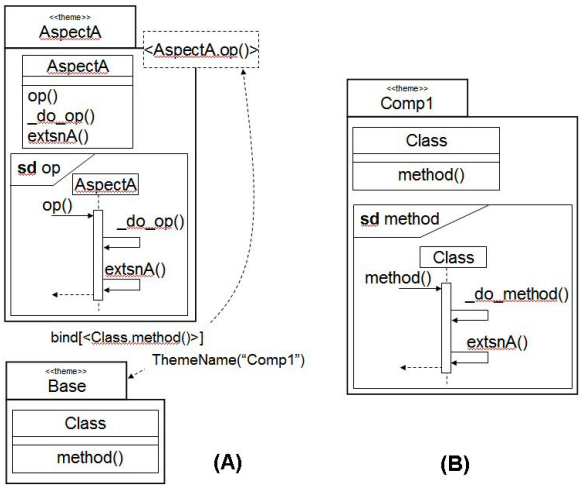

Figure 6. Theme/UML representation of `is also`

3.1 Aspect-Oriented Type Extension

Figure 6a illustrates the Theme/UML design of an aspect theme and a base theme. The aspect theme, called `AspectA`, is designed so that the crosscutting behaviour contained in the method `extsnA()` is executed after any method from the base system (represented in this example by `method()` in the base theme bound to the aspect). The result of composing the aspect with the

[1] A cover group contains one or more coverage objects.

base is shown in Figure 6b, where the crosscutting behaviour executes after the base method. This example shows how Theme/UML can be used to graphically represent *e*'s `is also` extension. The `is first` and `is only` extensions can be modelled in a similar fashion.

However, handling type extension is more problematic. One possible way of handling type extension is to have a base class that is extended to create a new class, e.g., we could create a base class called `dog` and a child class `pug` that is a specialisation of `dog`. However, in *e* the child class does not exist until it is generated at runtime, and it is not guaranteed that it will be generated, i.e., it would be possible to constrain the environment to only generate poodles and bulldogs, in which case although the possibility of a pug exists, it is never realised. It is difficult to graphically express this concept in Theme/UML due to its foundations in UML and the misalignment between the object-oriented concepts in UML and the dynamic, aspect-based module creation in *e*. An attempt can be made using sequence diagrams as they have the ability to represent object creation over time with lifelines. However, behaviour described with sequence diagrams become difficult to visualise and understand as designs grow and aspect-oriented extensions are added. Development of a domain-specific modelling language to explicitly address these issues (and the others mentioned throughout this paper) would be superior to the continued use of UML-based approaches to model *e*.

3.2 Language Dynamicity and Constraints

UML is best suited to modelling static languages, i.e., those in which all variable types are known at compile time. This raises questions about how Theme/UML models can cater for *e*'s dynamic language constructs. To model the dog example from listings 2 and 3 we could have a dog generator that creates dog structs, and so might have multiple copies that are created and destroyed during the program execution. However, we might also have a constraint that says `keep dog_kind in [poodle,pug]`, meaning that a bulldog can never be created although the code for it exists. Formal constraints can be added to variables via the Object Constraint Language (OCL), but we found we could not model constraints that are modified via aspect extension.

```
type bark_type : [yelp, growl, howl];
type dog_kind : [poodle, pug, bulldog];

struct dog {
   bark : bark_type;
   dog_kind : dog_kind;
   keep soft bark_type in [yelp,growl];
};

extend poodle dog {
   keep bark_type=yelp;
};
```

Listing 6. Aspect-oriented code extension

Listing 6 illustrates how constraints can be extended via aspects. The `soft` constraint restricts the `bark_type` to only be yelp and growl. Soft constraints can be overridden at any time, although the constraint engine will try to satisfy soft constraints if they overlap with hard constraints. For the extended `poodle dog`, the `bark_type` is constrained further via an aspect. Graphical representation of this form of constraint extension is not catered for in UML.

Furthermore, different constraints can interact to reduce the allowed state space, and may even contradict. As constraints can be statically analysed, it should be possible to highlight possible contradictions within UML by indicating how each constraint reduces the range of allowed possible values for the variable in question. A constraint set might allow a bulldog to be generated at certain times, but not at others, but we found it is not possible to describe temporally conditional existence with UML.

3.3 Timed Behaviour

The next question is how to represent a) timed methods that are tied to an event or clock to synchronise their execution b) temporal expressions that can be events or temporal checks and c) coverage objects. We found that it is possible to treat all of these as different types of method, as they have to be instantiated within a unit or struct. An event is a method that raises a flag when the event is emitted or when a sequence of other events is observed. A temporal expression is a piece of Boolean logic that evaluates to true or false and again raises a flag – a false for the entire expression is observed as a fail, whereas a true is observed as a pass. Coverage objects are also methods that increment counters for coverage purposes.

However, the main problem with modelling time constructs is that UML does not cater for real-time by default, making it difficult to model the difference between a standard method, e.g., `method()`, and a timed method, e.g., `method()@clock`, both of which are treated differently in *e*. Although a number of real-time extensions for UML have been proposed [5, 6], Theme/UML does not currently support design of real-time behaviour and therefore it was not possible to adequately model *e*'s time constructs. However, even in the case that a real-time profile for UML had been adopted, we would still have faced challenges when attempting to model *e*'s aspect-oriented behaviour. For example, the temporal behaviour of a method can be modified via an aspect by either changing the event to which the method is synchronised or by changing the code within a method. Additionally, *e* enforces different extension rules depending on which construct is being extended. For example, `is only` extension is not allowed for cover objects, but is allowed for methods. These behaviours are not supported by standard real-time profiles for UML.

3.4 Tool Support

From a more practical perspective, an additional problem is that aspect-oriented modelling is not currently supported in any of the commercial modelling tools that Infineon might use. The Theme/UML diagram presented in Figure 6 was created manually in a graphics package. Consequently it is not possible to load up an entire *e* environment and visualise it graphically with a tool that provides the quality guarantees that a large organisation requires. Tool support for Theme/UML has been developed recently and is described by Carton et al. in [6]. However, this tool supports standard Theme/UML and therefore cannot fully model *e* systems due to the issues discussed throughout this section.

4. REQUIREMENTS FOR ASPECT-ORIENTED MODELLING APPROACHES

In this section we present some requirements for aspect-oriented modelling approaches based on our experience with attempting to

model *e* using aspect-oriented extensions to UML. These are requirements that must be addressed if an aspect-oriented modelling approach wishes to support representation of hardware verification testbenches.

4.1 Visualising Aspect Interaction

The standard aspect development approach at Infineon involves endeavouring to group all related behaviour for a given aspect into one file, and creating a new file for each new extension to the original aspect (especially when reusing an existing piece of verification intellectual property). With a number of files under consideration, very different behaviour can occur depending on the order in which the files are loaded. Consider the example in Listing 7. If file A is loaded, followed by file B and file C, and `method()` is invoked, the message "I am C" is printed. If file A is loaded, followed by file C and file B, and `method()` is then invoked, two messages are printed, "I am C" followed by "I am B". The order in which sub-types are defined, extended via aspects and resolved at runtime also impacts program execution.

```
// file A:
method() is {
    print "I am A";
};
// file B:
method() is also {
    print "I am B";
};
// file C:
method() is only {
    print "I am C";
};
```

Listing 7. Extended *e* code in three files

The effect of load order is extremely important, and leads to one of the biggest headaches with aspect-oriented programming in *e* – debugging. When trying to find the exact piece of code to modify, the developer has to bear in mind that it may have been extended via aspects in several places, and previous extensions might have been completely overridden by new `is only` extensions. Consequently, carefully inserted debug messages can be completely ignored due to an unknown override.

We believe that in order to address this issue, a means of visualising an aspect-oriented system and how the aspects interact and resolve themselves is required. The modelling technique could be three-dimensional, with a time axis showing the effect of loading in different aspects at different times. This would allow the user to view the interaction effects of different aspect extensions as they are applied, and thereby get a greater understanding of the effects of load order on the resulting system.

4.2 Legacy System Support

Any modelling language adopted by Infineon must be able to support representation of legacy systems. We have been using the *e* language for eight years and our most significant problems concern support of our legacy systems. *e*VCs are typically written by small teams who may no longer be in the company or might not be available when necessary. These *e*VCs are imported into larger environments to test modules such as memory controllers. A memory controller is connected to a bus driven by an *e*VC, and to various memory models connected via call-backs to the *e* testbench so that both end-to-end scoreboarding and functional correctness assessment can be carried out. A typical development process is one in which a new memory class has to be supported, with new features added to the memory controller. The new memory model therefore has to be incorporated into the testbench, existing checks have to be updated, new checks have to be added, and if the bus protocol has been extended (perhaps by adding sideband signals), the *e*VC must be extended. If the original authors are not present for consultation, undertaking this work is a significant challenge, particularly in the absence of an adequate means of viewing and reasoning about the existing system. In our opinion, the lack of visualisation software for graphically representing aspect-oriented systems is the single largest problem facing the adoption of aspect-orientation in the hardware verification community.

4.3 Visualising Dynamic Behaviour

Any modelling approach adopted by Infineon has to understand the dynamic nature of *e*, as structs can be generated and destroyed during a simulation. Struct generation occurs in two ways - start-time generation and runtime generation. In start-time generation, *e* generates pseudo-random values to apply to all variables within the system, satisfying all constraints, and creating all units and structs that should exist at time 0. In runtime generation, in-line generate statements are used within methods to create new structs.

```
unit kennel {
 !list_of dogs : list of dog_struct;
// ! Means do not generate at start
//time, leave it null.
 method()@clk is{
   while TRUE {
      wait cycle;
      gen dog keeping {it.kind in
        [pug,poodle]};
      list_of_dogs.add(dog);
       if list_of_dogs.size()>4 {
         list_of_dogs.pop0()};
      // pop0() removes the bottom-most
      // list element
    };
  };
};
```

Listing 8. Dynamic struct generation

In Listing 8, `method()` runs, and every cycle it loops and re-runs. Therefore, at every clock cycle a new dog is generated by the `gen` method that constrains the dog kind to be one of two types (although the dog may have other types available, as in Listing 2). This new dog is then added to the `list_of_dogs`, and if the new list size is greater than 4, the dog at the bottom of the list is removed, i.e., the struct is effectively deleted. We therefore need to be able to model a creation point within the code and assign constraints to the creation point to restrict the range of structs that need to be modelled as possibilities at the creation point. There also needs to be some way of indicating the destruction of structs, and when they are removed from the system. However, there is nothing stopping the user extending inbuilt methods such as `pop0()` using an `is only` extension to change the behaviour of the system, so that it no longer deletes structs but instead manipulates them or even generates new structs. Essentially, a modelling approach cannot rely on keywords for inbuilt methods to establish what is happening but needs some way of visualising both the dynamic creation and

destruction of modules and the way in which this behaviour is affected by aspect extensions.

4.4 Modelling Time

The *e* language supports the concept of time (as discussed in Section 3.3 and illustrated in Listing 8). Therefore, any modelling approach used to represent *e* systems has to understand the concept of real-time and time consumption. In addition to emitting events, a time-consuming method can wait for an event, sync to an event, or wait a number of clock cycles. This can lead to the situation in Listing 8 where a perpetually-running method is set off. If `method()` was not synchronised to a clock, a 0-cycle delay loop is set off, causing the simulation to hang (the Specman simulator would execute `method()` forever without advancing time, and the system would lock). A modelling environment should be able to detect the difference between the two types of method, and highlight such a scenario as a potential problem.

An additional problem with modeling time is that the temporal behaviour of a method can be completely changed via aspect extension. Time consuming methods are synchronised to clocking events, and these events can be extended via aspects as discussed in Section 2. For example, an aspect could be used to change the behaviour described in Listing 8 by altering both the amount of time consumed when performing the `wait cycle` and the frequency with which `method()` repeats. In addition, the method itself can be extended using `is also`, `is only` or `is first` and these extensions can specify temporal behavior.

We also believe that a means to graphically represent time in *e* would be of significant benefit when handling temporal expressions (TEs). TEs are used to describe and check the behavior of the DUT over time, and are associated with sampling events. These events can themselves be extended, as illustrated in Listing 9.

```
event clk is rise('clk)@sim;
expect TE1 => TE2 @clk;
event clk is only fall('clk)@sim;
```

Listing 9. Aspect-oriented temporal extension

The `expect` statement is a runtime check that is triggered every time the `clk` event is true, i.e., on the rising edge of `clk` in the DUT. The expect statements looks for `TE1` being true on one rising edge of `clk`, and `TE2` being true on the next rising edge of `clk`. The `is only` extension modifies the `clk` event so that it is now true on every falling edge of `clk`, which implicitly changes when the `expect` statement is triggered. The temporal check in the `expect` statement does not have to be over one clock cycle, it could be over several cycles and involve several events or TEs, and each one can be extended using aspects.

4.5 Integration With Other Languages

Any approach for modelling hardware verification environments must be able to interface with concurrent High-level Design Languages (HDLs) such as Verilog and VHDL, as well as hardware-specific extensions of conventional languages such as SystemC that support hardware modelling. All hardware verification environments model some type of system behaviour, and sample the DUT to check for correct operation. The DUT should ideally be modelled as well, but generally it is not. In addition, the ideal verification development process includes development of an early C model of the DUT which is later replaced with the actual HDL model. Therefore, the modelling approach used to represent an aspect-oriented hardware verification environment needs to be able to reference external components that might be implemented in different languages and that might change over time.

5. Summary

In this paper we have discussed our experience with using an aspect-oriented modelling language to model hardware verification testbenches written in the *e* programming language. The dynamic, temporal nature of *e* meant that adequately modelling *e* testbenches was beyond the scope of Theme/UML, primarily due to its foundations in UML. Based on these findings we have proposed a number of requirements that should be addressed by aspect-oriented modelling approaches if they aspire to modelling hardware verification environments.

Acknowledgements

This work was supported, in part, by the Science Foundation of Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] Infineon Technologies. Online; accessed 17 September 2008. `http://www.infineon.com/`

[2] Clarke, S. and Baniassad, E. Aspect-Oriented Analysis and Design: The Theme Approach, 1st ed. Addison-Wesley, 2005.

[3] Cadence Design Systems. Online; accessed 17 September 2008. `http://www.cadence.com/`

[4] The *e* Functional Verification Language Working Group. Online; accessed 17 September 2008. `http://www.ieee1647.org`

[5] Douglass, B.P. Real-Time UML: Advances in the UML for Real-Time Systems, 3rd ed. Addison-Wesley, 2004.

[6] Gherbi, A. and Khendek, F. UML Profiles for Real-Time Systems and their Applications. Journal of Object Technology 5, 4 (May – June), pp. 149 – 169.

[7] Carton, A., Driver, C., Jackson, A. and Clarke, S. Model-Driven Theme/UML. To appear in Transactions on Aspect-Oriented Software Development, Springer 2009.